JRun Samples Guide

JRun 3.0 for Windows®, UNIX™, and Linux™

Allaire Corporation

Copyright Notice

Part number: AA-EJSMP-RK

Contents

CHAPTER 1

Introduction

The samples provided with JRun illustrate the various ways that the server can be used. They include servlet samples and EJB samples.

Contents

About JRun Samples

JRun includes a full suite of samples that show you how to use servlet technologies (servlets, JSP pages, and custom tags) and EJBs. Running these samples and examining the associated source code can help you get started with JRun.

To run the samples you must have completed all of the installation steps. If you have not already done so, review the *JRun Setup Guide* before attempting to run any samples.

Samples for Servlet Technologies

JRun includes samples for all types of servlet technologies. This includes JSP samples, custom tag library samples, and servlet samples. You access these samples by clicking Example Applications from the JMC Welcome page.

JSP sample overview

JRun provides the following JSP samples:

- Hello world sample
- Color size bean sample
- JavaScript sample
- QueryString sample
- HTML form example

You can find JSP samples in Chapter 2.

Hello world sample

Displays `Hello World` in gradually increasing and decreasing point sizes.

Color size bean sample

Shows you to how to use methods and properties in the `ColorSizeBean` JavaBean.

JavaScript sample

Shows you how to use JavaScript (instead of Java) in a JSP page.

QueryString sample

Shows you how to use the `request.getParameter` method to access text from the query string.

HTML form example

Shows you how to code a form that uses JSP.

Custom tag library sample overview

The JSP 1.1 specification includes a JSP extension mechanism known as tag libraries. A tag library defines a set of custom tags (also known as actions) that encapsulate certain types of functionality. JRun includes a custom tag library and the JRun examples include the following samples of using this tag library:

- Query and QueryParam sample
- Form, Input, and Select sample
- Servlet and ServletParam sample
- Param sample
- ForEach sample

You can find JSP samples in Chapter 3.

Query and QueryParam sample

Show you how to issue SQL statements using the `query` and `queryparam` tags.

Form, Input, and Select sample

Show you how to enhance interactivity by using the `form`, `input`, and `select` tags.

Servlet and ServletParam sample

Show you how to invoke servlets using the `servlet` tag and pass parameters using the `servletparam` tag.

Param sample

Shows you how to declare a JSP scripting variable using the `param` tag.

ForEach sample

Shows you how to code a loop using the `foreach` tag.

Servlet sample overview

JRun provides the following servlet samples:

- SimpleServlet sample
- DateServlet sample

- CounterServlet sample
- SnoopServlet sample

You can find servlet samples in Chapter 4.

SimpleServlet sample

Shows you how to set the content type and return HTML that includes a simple text string.

DateServlet sample

Shows you how to display the current date/time and how to refresh the page automatically.

CounterServlet sample

Shows you how to use cookies to maintain a page hit counter.

SnoopServlet sample

Shows you how to retrieve and display servlet and environment information.

EJB Samples

JRun includes samples that show many different types of EJB usage. You access these samples by reading the associated instructions and executing sample-specific `make` files.

EJB sample overview

JRun provides the following EJB samples:

- Sample 1 - Getting Started
- Sample 2 - Bean Managed Persistence
- Sample 3 - Container Managed Persistence
- Sample 4 - Transactions
- Sample 5 - Object Management
- Sample 6 - Messaging
- Sample 7 - Advanced Beans
- Sample 9 - Servlets
- Sample 10 - JDK 1.1 Clients

Sample 1 - Getting Started

Sample 1 (described in Chapter 5) demonstrates simple communication between the server and clients. Functionally the sample provides the ability to adjust a balance by saving or spending, with the balance persisted over time and represented as an entity bean. Users are authenticated and authorized to take on roles while calling methods. The file, `instance.store` is used for persistence.

The Sample 1 documentation provides a detailed description of using the `make` files to run an EJB sample. Additionally, this sample includes beans that illustrate how to extend functionality.

Sample 1b uses custom beans for authentication.

Sample 2 - Bean Managed Persistence

Sample 2 (described in Chapter 6) demonstrates using bean managed persistence to a relational database. This sample uses the same functionality as Sample 1. Access to a relational database is required to run this sample.

Sample 3 - Container Managed Persistence

Sample 3 (described in Chapter 7) demonstrates using container managed persistence to a relational database. This sample uses the same functionality as Sample 1.

Sample 4 - Transactions

Sample 4 (described in Chapter 8) illustrates distributed 2-phase commit transaction management across multiple server instances. This sample uses slightly modified functionality based on Sample 1.

Sample 5 - Object Management

Sample 5 (described in Chapter 9) illustrates distributed garbage collection capabilities where no-longer referenced entity objects are released and garbage collected. The sample also demonstrates creating large numbers of objects and returning them as collections to a client as well as creating custom RMI objects.

Sample 6 - Messaging

Sample 6 (described in Chapter 10)demonstrates Java Message Service (JMS) support with both point-to-point (message queueing) and publish/subscribe (broadcast) mechanisms. It also illustrates JMS integration with EJB.

Sample 7 - Advanced Beans

Sample 7 (described in Chapter 11) demonstrates the use of different types of beans (entity, stateful session, and stateless session) working together to solve a complex business problem. This sample demonstrates the handling of deadlock exceptions and the use of autocallers. Functionally the sample simulates the issuance of loans by banks to customers. The number of banks, along with the interest and income rates, are set in the `deploy.properties` file. The number of customers is defined at runtime with command line arguments.

Sample 7b demonstrates the same functionality as Sample 7a but uses bean managed persistence through a relational database.

Sample 7c demonstrates the same functionality as Sample 7a but uses container managed persistence through a relational database.

Note There is no Sample 8.

Sample 9 - Servlets

Sample 9a (described in Chapter 12) illustrates using servlets and JSP with the EJB engine.

Sample 10 - JDK 1.1 Clients

Sample 10a (described in Chapter 13) provides instructions on using the EJB engine with JDK 1.1 clients.

Additional EJB information

This discussion contains additional information that will help you understand the EJB samples.

Notes

The EJB samples run the EJB engine in standalone mode. This allows you to view bean processing in a console window. The EJB engine runs using the port settings of the JRun default server so the JRun default server cannot be running while you are executing the EJB samples. For information on running the EJB samples with the JRun default server, see “Using Make Standalone” on page 72.

References to a ‘bean’ or ‘beans’ should be interpreted as ‘Enterprise JavaBeans’.

The `instance.store` flat-file database is used by various samples. You should clear the `instance.store` after running each sample to avoid picking up data from a prior sample. To clear the `instance.store`, go to the `/jrun/servers/default/runtime` directory and delete `instance.store`.

Notation

For the purpose of simplifying the samples, the notation `/jrun` is used to generically represent the JRun installation directory. Where you see `/jrun` simply replace it with the appropriate path for your installation.

Before running any samples, you must set the `JRUN_HOME` environment variable to the JRun installation directory. References to `/jrun` and `JRUN_HOME` are synonymous.

Note The difference between `/jrun` and `JRUN_HOME` is this: `/jrun` is a generic reference to the JRun installation directory; `JRUN_HOME` is a variable used in the `make` (and `makew`, described below) files and this variable must point to the JRun installation directory.

Commands

EJB samples are driven by `make` files that compile, package, and execute the files necessary to create the proper sample execution environment. There are a few basic `make` commands, which are outlined in Chapter 5 and discussed in detail in Chapter 14. Batch files, similar to the `make` files, are provided for Windows users. Instead of typing the `make` command, Windows users type the `makew` command, which runs the batch files. Remember to set the `JRUN_HOME` environment variable before running `make` or `makew`.

Additional Samples

Periodically check the JRun area of the Allaire DevCenter (`http://www.allaire.com/developer/jrunreferencedesk`) for additional samples, including the Tack2 sample application, which demonstrates a variety of servlet technologies and techniques in the context of a database driven e-commerce site.

CHAPTER 2

JSP Samples

Contents

JSP Sample Overview

JRun includes JSP samples to help you understand the basic concepts of JSP coding.

To get the most out of these samples, you should run them, review the source code, and then run them again. To run the JSP samples, start the JMC and click Example Applications from the Welcome page. You can find source code for the JSP samples in `/jrun/servers/default/demo-app/jsp`.

Hello World

Description

Displays `Hello World` in gradually increasing and decreasing point sizes.

File name

`hello.jsp`

What to look for

Review the `for` loop and the use of JSP expressions to control increasing and decreasing point size.

Color Size Bean

Description

Shows you to how to use methods and properties in the `ColorSizeBean` JavaBean.

File name

`colorsize.jsp`

What to look for

Review the `jsp:useBean` and `jsp:setProperty` statements. Also review the usage of calls to methods in `ColorSizeBean`. You can find the source code to `ColorSizeBean.java` in `/jrun/servers/default/demo-app/WEB-INF/classes`.

JavaScript Example

Description

Shows you how to use JavaScript (instead of Java) in a JSP page.

File name

`javascript.jsp`

What to look for

Note `language=javascript` in the `page` directive. Also note the use of the JavaScript `Date` object and the call to the `Date` object's `toString` method.

QueryString Example

Description

Shows you how to use the `request.getParameter` method to access text from the query string.

File name

`qstring.jsp`

What to look for

Review how the calls to `request.getParameter` return size and color from the query string.

HTML Form Example

Description

Shows you how to code a form that uses JSP.

File name

`form.jsp`

What to look for

Review the usage of `request.getParameter` and `request.getParameterValues` to access form data,

CHAPTER 3

Tag Library Samples

Contents

Tag Library Sample Overview

The JSP 1.1 specification includes a JSP extension mechanism known as tag libraries. Each tag library defines a set of custom tags (also known as actions) that encapsulate certain types of functionality. JRun includes a sample custom tag library, which you can use in JSP pages to provide data access, form validation, servlet access, and other types of functionality.

JRun includes tag library samples to help you understand how to use custom tags.

To get the most out of these samples, you should first review tag library documentation, which is available from the JRun documentation page. Once you are familiar with tag library usage, run the examples and review the JSP source code. To run the tag library samples, start the JMC and click Example Applications from the Welcome page. You can find source code for the tag library samples in `/jrun/servers/default/demo-app/taglib`.

For information on developing tag libraries, refer to the Custom Tags and Tag Libraries chapter in *Developing Applications with JRun*.

Query and QueryParam

Description

These samples show how to access a SQL database using the `query` and `queryparam` tags.

Note To use the `query` tag, you must have installed a JDBC driver and used the JMC to add your JDBC classpath to {app.classpath}.

File name

`query.jsp` and `queryparam.jsp`

What to look for

Review the syntax and sample page. By default, these samples are display-only. If you have a JDBC data source available and a JDBC driver installed, add it to the application classpath using the JMC, modify and uncomment the query tag, and rerun the sample.

Form, Input, and Select

Description

These samples show you how to enhance interactivity by using the `form`, `input`, and `select` tags.

File name

`form.jsp`, `input.jsp`, and `select.jsp`

What to look for

These tags allow you to create HTML forms with built-in client-side JavaScript validation. Use the input and select tags to declare validation and error criteria. The tags automatically generate JavaScript to perform the requested functionality. When you display these samples, view the HTML source to see the automatically generated JavaScript.

Servlet and ServletParam

Description

These samples show you how to invoke servlets and pass parameters.

File name

`servlet.jsp` and `servletparam.jsp`

What to look for

Note how the `servletparam` tag is used to pass parameters.

Param

Description

This samples shows you how to declare a JSP scripting variable.

File name

`param.jsp`

What to look for

Review the use of the `type` attribute, which specifies the data type, and the `default` attribute, which specifies a default value.

ForEach

Description

This sample shows you how to code a loop.

File name

`foreach.jsp`

What to look for

Use the `foreach` tag to loop over an `Enumeration`.

CHAPTER 4

Servlet Samples

Contents

Sample Servlet Overview

JRun includes servlet samples to help you understand the basics of coding with the servlet API.

To get the most out of these samples, you should run them, review the source code, and then run them again. To run the servlet samples, start the JMC and click Example Applications from the Welcome page. You can find source code for the servlet samples in `/jrun/servers/default/demo-app/WEB-INF/classes`.

Note All JRun sample servlets extend the `JRunDemoServlet` class, which enables the sample servlets to display a consistent look-and-feel. In particular, note that the sample servlets call the `generateDemoPageStart` and `generateDemoPageEnd` methods, which are defined in `JRunDemoServlet`.

JRunDemoServlet

Description

Abstract class used as a base class for all JRun sample servlets. Sample servlets extend this class and call the `generateDemoPageStart` and `generateDemoPageEnd` methods to build HTML for the beginning and ending of each page.

File name

`JRunDemoServlet.java`

What to look for

Review the `generateDemoPageStart` and `generateDemoPageEnd` methods. Note how they return HTML through the `out` variable. Also note how the `out.println` methods integrate quoted text and object variables (such as `ROW_ALT_COLOR` and `TITLE_COLOR`).

SimpleServlet

Description

Shows you how to set the content type and return HTML that includes a simple text string.

File name

`SimpleServlet.java`

What to look for

The servlet creates a `PrintWriter` through the `HttpServletResponse.getWriter` method. The servlet uses this object to return HTML to the browser. Also note that the servlet sets the content type to `text/html` via the `HttpServletResponse.setContentType` method.

DateServlet

Description

Shows you how to display the current date/time and how to refresh the page automatically.

File name

`DateServlet.java`

What to look for

The servlet creates a `Date` object and uses the `toString` method to return the current date and time as a `String`. Also note that the servlet uses a URL parameter (named `mode`) to implement a simple auto-refresh feature.

CounterServlet

Description

Shows you how to use cookies to maintain a page hit counter.

File name

`CounterServlet.java`

What to look for

The servlet retrieves all cookies into an array by calling the `HttpServletRequest.getCookies` method. It then loops through the array looking for a cookie named `counter`. If the `counter` cookie is not found, the servlet establishes the cookie and sets its value to 1. If the `counter` cookie is found, the servlet displays the current value and increments the cookie.

SnoopServlet

Description

Shows you how to retrieve and display servlet and environment information.

File name

`SnoopServlet.java`

What to look for

The servlet implements the following private methods to retrieve servlet and environment data:

- `getInitParameterData`
- `getContextParameterData`
- `getAttributeData`
- `getSessionData`
- `getRequestParameterData`
- `getRequestParametersData`
- `getHeaderData`
- `getCookieData`
- `getRequestData`

Review these methods to see how the servlet API methods are called to retrieve data.

This servlet also features collapsible table display, which you can review in the `makeTableEntry` method.

CHAPTER 5

Getting Started with EJB Samples

Contents

Overview

There are a few basic commands that are used to run every sample. These commands are discussed in detail in Sample 1a. The notation `/jrun` has been used here to generically represent the JRun installation directory. Where you see `/jrun`, replace it with the appropriate path for your installation. For the purpose of simplifying the samples, it is assumed that JRun was installed under `/opt/jrun` for UNIX and Linux, or `C:\Program Files\Allaire\JRun` for Windows.

It is recommended that you run Sample 1a each time you reinstall JRun. This is a simple and quick test for confirming that your environment is set up correctly.

Before You Begin

Before running any of the EJB samples, review your system configuration to ensure that required resources are available, as follows:

- Your system must be running JDK 1.2. The EJB engine will not work with JDK 1.1.
- The directory that contains your Java compiler must be on the system path. For example, `jdk1.2.2/bin`. If this directory is not on the system path, the `make jars` command will fail.
- Ensure that you have defined an `JRUN_HOME` environment variable and that this variable points to the JRun root.

Sample 1a - Simple Beans and Security

Sample 1a provides the ability to adjust a balance by saving or spending. The balance is persisted over time and is represented as an entity bean. Users are authenticated and authorized to take on roles while calling methods. The sample accesses users and roles defined in the `deploy.properties` file. The default `instance.store` is used for persistence.

The `BalanceBean.java` implements the business logic for updating the balance. Review the `Balance.properties` file to see how properties for the Balance bean are set. The `deploy.properties` file contains the server name along with users and their respective roles. Also take a look at the `manifest` file; beans must be listed in the `manifest` file in order to be deployed.

To see how the client side works, start with `EjbClient.java`. Notice there are no manifest or property files required for the client. The client is authenticated through JNDI.

To review the complete set of JavaDocs for this sample go to `/jrun/samples/sample1a/docs`.

Running the samples

Setting the host name

The EJB engine must know where the class server is located. This is done by setting the host property. Go to /jrun/samples/sample1a and open the deploy.properties file using any text editor. Notice the ejipt.classServer.host property is currently set to localhost. Change this setting to either the host name or IP address of the host. If you are running the server and clients locally, this property can be set either to your machine name or can remain localhost. Be sure to save any changes. If you leave this property unspecified, JRun will default it to the name of the host where the server is running.

Note that all clients must be able access the server with the host name or IP address specified in the ejipt.classServer.host property. This is particularly important when going through firewalls and across networks.

Open a shell (bash shell)

Now open a command prompt window and enter the following commands (opt/jrun is the default JRun install directory on UNIX):

```
> bash
bash$ export JRUN_HOME=/opt/jrun
bash$ cd /opt/jrun/samples/sample1a
```

By default, these discussions use the Bourne-Again SHell (bash). The bash command creates a bash shell for running the make files. The export command sets the environment variable JRUN_HOME to the directory where JRun is located — in this case /opt/jrun.

Be sure to use the forward slash (/) as the separator when working in the bash shell. You can use another shell if you prefer. However, the bash shell is required for running the make files verbatim.

Open a shell (DOS shell)

Under Windows, you can also use a DOS prompt window and enter the following commands:

```
set JRUN_HOME=c:\Program Files\Allaire\JRun
cd "c:\Program Files\Allaire\JRun\samples\sample1a"
```

The set command sets the environment variable JRUN_HOME to the directory where JRun is located — in this case c:\Program Files\Allaire\JRun.

Note The remaining examples in this manual use the bash shell. Remember that whenever the sample specifies make, Windows users should type makew.

Create the bean jars

You are now ready to create the jar files for the beans and the client. Enter the following command (remember to use makew on Windows):

```
bash$ make jars
```

The make jars command compiles the EJB source files, creates the sample1a_ejb.jar bean .jar file, and copies it to the /jrun/servers/default/deploy directory. It also creates sample1a_client.jar for the client and copies it to the /jrun/samples/sample1a directory.

Deploy the beans

Now you are ready to deploy the beans. Enter the following command (remember to use makew on Windows):

```
bash$ make deploy
```

The Deploy tool generates the implementations of the home and object interfaces using the sample1_ejb.jar created in the make jars step. The resulting ejipt_objects.jar is placed into the deploy directory. This step also generates the stub classes and again places the resulting ejipt_exports.jar into the deploy directory. Make deploy then copies the deploy.properties to the deploy directory and uses it as a base to create the runtime.properties file.

Note The make deploy command copies the sample-specific deploy.properties file from the sample's directory to the /jrun/default/deploy directory. This technique is used to ensure the integrity of each individual example. However, once work begins on your own EJBs, you will only work with the deploy.properties file in the /deploy directory.

The Deploy tool defaults to using the JDK 1.2 compiler to compile the generated classes. You can use a different compiler, such as Jikes, by setting the various ejipt.javac.* properties in the deploy.properties file.

Start the EJB engine

Once the beans have been deployed, you start the EJB engine in stand-alone mode. Enter the following command (remember to use makew on Windows):

```
bash$ make standalone
```

When the process completes, you should see the EJB engine's command prompt:

```
Server is running (type h[elp]<ENTER> for help on commands)
>
```

The 'make standalone' command starts the EJB engine using the jar files in the deploy directory. The .jar and .properties files in the deploy directory are copied to the runtime directory. The EJB engine is now ready to accept client requests.

Note The `make standalone` command starts the EJB engine using the directories and port settings of the JRun default server. You must stop the JRun default server before issuing `make standalone`.

Start the client

To start the client, open another command prompt window and enter the following commands (remember to replace `/jrun` with your JRun install directory and to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

You will now see the client login window. Refer to How to Use Sample 1a for a description of how to use the sample. To stop the client, press the Exit button on the login window.

Stop the EJB engine

Once you are finished with the sample you will want to stop the EJB engine. To stop the EJB engine, type `q` and press Enter. You will see output similar to the following:

```
>q
Server stopped
```

How to Use Sample 1a

Once you have both the server and the client running, you can proceed with running the sample. Enter your host name in the client's *Server* text field, this will be the same value you set for the `ejipt.classServer.host` property in the `deploy.properties`. Next enter “saver1” for the user and “pass” for the password and press the *Login* button.

You will see a new screen containing fields for *Amount* and *Repea*t. Enter a value in the *Amount* field and the number of repetitions in the *Repeat* field. Press the *Save* button. You will see the balance changing in the server window. Try pressing the *Spend* button; since you logged in as a saver you are not allowed to spend.

Now press the *Logout* button and login again, this time as “spender1/pass”. Since you are now a spender, you will not be allowed to save. Logging in as “chief/pass” allows you to save as well as spend. If you look in the `/jrun/samples/sample1a/deploy.properties` file you will see the following entries defining users and roles:

```
ejipt.users=spender1:pass;spender2:pass;saver1:pass;saver2:pass;
chief:pass
ejipt.roles=spender:spender1,spender2,chief;saver:saver1,saver2,chief
```

You can change the user names, passwords, and roles. Once you make any modifications, be sure to run the following commands to try out your changes (remember to use `makew` on Windows):

```
bash$ make deploy
bash$ make standalone
bash$ make run
```

Then restart the client by opening another command prompt window and entering the following 4 commands (remember to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

If you stop the server and then restart it again, you will notice that your balance has been persisted. JRun is using its `instance.store` for persisting objects. If you would like to start the server with no balance, then go to the `JRUN_HOME/servers/default/runtime` directory and delete `instance.store`. Any data stored will, of course, be lost if you do this.

Multiple clients

To create additional clients, open another command prompt and enter the following 4 commands (remember to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

Sample 1a Usage Scenarios

The following sections illustrate the EJB engine running in embedded mode, instantiated in a standard Java class, in-proc, subclassed, and in debug mode using the Java Debugger.

Embedding the EJB engine

The EJB engine can be embedded in your application and instantiated just as any other Java class resulting in in-proc calls between your application and the EJB engine.

Note This is an advanced topic. You should run all EJB samples before attempting this sample. For more information on embedding the EJB engine, refer to the *JRun Advanced Configuration Guide*, available from the Allaire DevCenter.

To see how this can be done, change to the /jrun/samples/sample1a directory. You will see a file named Server.java which embeds the EJB engine. Open Server.java in any editor. You will see the following snippet, beginning on line 17:

```
Ejipt.prepareEnvironment(true);
Ejipt.prepareProperties(null);
final Ejipt ejipt = new Ejipt(true);
ejipt.start();
ejipt.export(0);
```

The Ejipt.prepareEnvironment(true); statement copies the required files, including stubs and properties, from the /deploy subdirectory to the /runtime subdirectory. The Ejipt.prepareProperties(null); statement then loads in properties from the various properties files. The final Ejipt ejipt = new Ejipt(true); creates an instance of the EJB engine. The ejipt.start(); statement loads the remote and home objects and the ejipt.export(0); statement exports the server. To run Sample 1a using Server, enter the following commands:

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make jars
bash$ make deploy
bash$ java -Djava.security.policy=/jrun/jrun.policy
    -Dejipt.home=/jrun -classpath “.;/jrun/lib/ejipt.jar” Server
```

Now to start the client, open another command prompt window and enter the following commands (remember to use makew on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

The client window will appear. Enter the server name, user id and password to login, then enter some transactions. You will see server output appear in the command prompt window where you started Server.

Server can be viewed as a shell that runs the EJB engine in standalone mode. It’s capabilities can be expanded to provide custom server functionality or end-to-end application solutions.

Subclassing the EJB engine

The allaire.ejipt.Ejipt class can be subclassed by a custom server with selected methods overridden. Again the EJB engine is running in standalone mode. To see this illustrated, go to the /jrun/samples/sample1a directory. You will see a file named CustomServer.java. Open the file in any editor. Just as in Server.java, you will see the preparation steps for starting the server.

Also notice that CustomServer.java includes the CustomEjipt class, which extends allaire.ejipt.Ejipt and that CustomEjipt overrides the logMessage and logWarning methods. Any output will be prefixed with either “Custom Message:” or “Custom Warning;”. These methods can be overridden to write to a log file or database.

Note This is an advanced topic. You should run all EJB samples before attempting this sample. For more information on subclassing the EJB engine, refer to the *JRun Advanced Configuration Guide*.

To run Sample 1a with `CustomServer` enter the following commands (remember to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make jars
bash$ make deploy
bash$ java -Djava.security.policy=/jrun/jrun.policy -Dejipt.home=/jrun
    -classpath ".;/jrun/lib/ejipt.jar" CustomServer
```

To start the client open a second command prompt window and enter the following commands (remember to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

Debug mode

You can run both `Server` and `CustomServer` in the Java Debugger.

Note This is an advanced topic. You should run all EJB samples before attempting this sample. For more information on debug mode, refer to the *JRun Advanced Configuration Guide*.

To run `Server` or `CustomServer` in the Java Debugger, enter the following command (notice "jdb" rather than "java"):

```
bash$ jdb -Djava.security.policy=/jrun/jrun.policy -Dejipt.home=/jrun
    -classpath ".;/jrun/lib/ejipt.jar" Server (or CustomServer)
```

At the prompt, enter the following commands:

```
> stop at ejbeans.BalanceBean:39
> run
```

Now open another command prompt window and start up the client (remember to use `makew` on Windows):

```
C:\> bash
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

Login and perform a transaction. You will see the server stop at line 40 of `BalanceBean`. Enter cont at the prompt to continue processing. To see additional commands, enter `help`. See the Java Debugger documentation for addition information and commands.

Running the EJB engine in this way allows you to step through your beans, making the debugging process very simple and straightforward.

Client updates

The behavior of Sample 1a can be changed so that the client receives the updated balance. The beans to do this are included in the sample1a directory. Change to the `/jrun/samples/sample1a/client` directory. You will see `EjbClient.java` and `EjbClient.java.new`. Rename `EjbClient.java` to `EjbClient.java.old`; then rename `EjbClient.java.new` to `EjbClient.java`.

Now go to the `/jrun/samples/sample1a/ejbeans` directory. Rename both `Balance.java` and `BalanceBean.java` to `old`. Then rename `Balance.java.new` and `BalanceBean.java.new` to `Balance.java` and `BalanceBean.java` respectively. To try out this new version enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1a
bash$ make run
```

You will see the balance transactions in the client window after issuing `make run` with the new version.

Dynamic bean loading

The behavior of Sample 1a can be changed so that it implements dynamic bean loading. To do this, open the `/jrun/samples/sample1a/ejbeans/BalanceBean.java` file and make a minor change, such as adding asterisks to the log messages:

```
public void save(final int value)
        throws RemoteException {
    _value += value;
    // Add asterisks.
    ResourceManager.getLogger().logMessage("***saving, balance is: " +
_value);
}

public void spend(final int value)
        throws RemoteException {
    _value -= value;
    // Add asterisks.
    ResourceManager.getLogger().logMessage("***spending, balance is: " +
_value);
}
```

Now compile `BalanceBean.java` and place it in the `classes` directory by entering the following command (remember to use `makew` on Windows):

```
bash$ make classes
```

Return to the server window and enter the following command:

```
> load
```

Finally, reissue save and spend requests on the client. You will see the modified messages, which indicate that the EJB engine is using the modified bean.

XML descriptor

The behavior of Sample 1a can be changed so that it accesses bean properties from an XML descriptor file instead of a bean properties file and the `default.properties` file. To run Sample 1a using a descriptor file, enter the following commands:

```
bash$ make xml_jars
bash$ make deploy
bash$ make standalone
```

When you use the `xml_jars` option, the EJB engine uses the `META-INF/ejb-jar.xml` file to access bean properties.

Sample 1b - Custom Authentication

In this sample, user identities are authenticated using custom beans. `LoginSessionBean`, `UserBean` and `RoleBean` have been added to the `ejbeans` directory. Review them to see how authentication and authorization are done. Notice that the user ids and passwords are no longer in the `deploy.properties` file.

To begin the demo, enter the following series of commands (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1b
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample1b
bash$ make run
```

When the client screen appears, login as described in Sample 1a. You may change the user names, passwords and roles by modifying `UserBean` and `RoleBean`. `LoginSessionBean`, `UserBean` and `RoleBean` can all be modified and extended to provide security and authentication services using rules from a database or from a directory service.

CHAPTER 6

Bean Managed Persistence

Contents

Overview

Sample 2 demonstrates bean managed persistence (BMP) using a relational database. Using BMP, your entity bean manages persistence by coding the appropriate logic (typically SQL statements) in certain callback methods.

This sample functions as in Sample 1. You will need access to a relational database to run Sample 2.

Sample 2a - Default Authentication

To begin this sample go to `/jrun/samples/sample2a/ejbeans` and take a moment to review `BalanceBean.java`. Notice the SQL statements in the `ejbPostCreate`, `ejbLoad`, and `ejbStore` methods. The EJB engine calls these methods at specific times in the bean's life cycle. and the SQL statements in these methods implement persistence for the bean. For more information on BMP, refer to the *Developing Applications with JRun* manual.

Now review the following line, which you will find in each of these methods:

```
final Connection connection = ResourceManager.getConnection("source1");
```

`ResourceManager` is a JRun class, found in the `allaire.ejipt` package, that allows you to simplify the database connection process. This line uses the `ResourceManager.getConnection` method to retrieve a connection to the database. Notice the reference to `source1`.

Now open the `deploy.properties` file to see how `source1` is defined. Notice the following entries:

```
ejipt.jdbcSources=source1
source1.ejipt.sourceURL=jdbc:odbc:sample
#source1.ejipt.sourceUser=xyz
#source1.ejipt.sourcePassword=pass
```

The `ejipt.jdbcSources` property defines the data sources that are available to the EJB engine through the `ResourceManager` class. It can contain multiple data sources in a comma separated list (for example, `ejipt.jdbcSources=source1, source2`). You can specify data source-specific properties by using the name specified in `jdbcSources` as a prefix.

The `source1.ejipt.sourceURL=jdbc:odbc:sample` property uses the standard Java JDBC convention for identifying the database through a URL. This example uses the JDBC/ODBC bridge to connect to a database named 'sample'.

If you are using something other than the jdbc/odbc bridge, you must change the properties to reflect that. In particular be sure to set the `@host` in `source1.ejipt.sourceURL` if you are using the Oracle driver. Now change 'sample' to the name of your database (for example, `jdbc:odbc:testdb`).

The next two properties are the account name and password for the database. In this example these two lines are commented out as a result of the # in the first column. If your database requires an account and password, remove the # from column one on each line and change 'xyz' and 'pass' to the appropriate values. You can define any number of data sources in this manner. Be sure to save your changes to `deploy.properties`.

To run Sample 2a create a table named "account" in your database. In the account table create the following columns:

Sample 2 Table Schema	
Column Name	**Column Type**
id (key)	INTEGER
value	INTEGER

To start the demo open a shell from the command prompt. If you are using a third-party JDBC driver, it must be installed on the server machine. Before starting the EJB engine you must enter the following command, being sure to provide the correct path for the driver:

```
bash$ export JDBC_DRIVERS=/path/driver_name
```

Be sure to set `JRUN_HOME` with the export command and change to the `/jrun/sample2a` directory. Enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window:

```
bash$ make run
```

Try some transactions and then check your database. You will see an entry containing an id and the current balance. The balance in the database will match the balance in the console window.

Sample 2b - Custom Authentication

Currently Sample 2b, uses the same user/role authentication as Sample 1b. This sample uses the same table that was created for Sample 2a.

To begin the demo, enter the following commands (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample2b
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make run
```

Again, you will see that the balance in your database equals the balance displayed by the console.

CHAPTER 7

Container Managed Persistence

Contents

Overview

Sample 3 demonstrates Container Managed Persistence (CMP) using a relational database. With CMP, the EJB engine manages entity bean persistence using the data source and SQL statements defined in the bean properties file or deployment descriptor.

This sample's functionality is the same as Sample 2. The option of modifying the sample to use the `instance.store` is also provided. See the description at the end of Sample 3a on how to do this.

Sample 3a has been tested with Oracle using the Thin JDBC driver. It has also been tested with SQLServer using the stock JDBC/ODBC bridge.

Sample 3a - Default Authentication

This sample uses the same database definition as Sample 2a. If you have not already done so, review Sample 2a for information on creating the required database tables.

Start by reviewing the `deploy.properties` file in `/jrun/samples/sample3a/`. Notice that there are properties defining the JDBC source,. These properties default to using the JDBC/ODBC bridge. Be sure to update the `ejipt.jdbcSources`, `ejipt.sourceURL`, `source1.ejipt.sourceUser`, and `source1.ejipt.sourcePassword` property settings as necessary for your environment. In particular be sure to set the `@host` in `source1.ejipt.sourceURL` if you are using the Oracle driver.

The `ejipt.logSQLRequests=true` property will cause all SQL calls to be displayed in the log window. This feature can be useful during debugging but should be set to false in production environments. Now review `Balance.properties` in `/jrun/samples/sample3a/ejbeans`, which contains the CMP properties.

The first property is `ejb.containerManagedFields=_id,_value`, telling the EJB engine that the `_id` and `_value` fields are to be managed by the container. The existence of this property notifies the EJB engine that there are container managed fields. This property must be set for the container to manage persistence for the bean.

The `ejipt.*SQL*` properties define how the EJB engine stores and retrieves the data in the bean. In this sample there are properties for `postCreate` (`ejipt.postCreateSQL`), `load` (`ejipt.loadSQL`), and `store` (`ejipt.storeSQL`). These properties are used by the container to properly manage the bean instance's state. There is no need for a `createSQL` since the argument is the primary key and requires no validation.

Note The `ejipt.postCreateSQL` property uses the `create_balance` database stored procedure to perform insert processing. The `/jrun/samples` directory includes files you can use to define this stored procedure for your DBMS. Use `sqlserver.sql` file to define the stored procedure for SQL Server and `oracle.sql` to define this stored procedure for Oracle.

A complete description of the SQL properties can be found in the container managed persistence discussion of *Developing Applications with JRun*. Now review

`BalanceBean.java` in `/jrun/samples/sample3a/ejbeans`. You will notice less code than in the corresponding Sample 2a `BalanceBean.java`. Notice in particular the `ejbPostCreate, ejbLoad`, and `ejbStore` methods. All SQL-related references have been removed.

Now run the sample by entering the following commands, replacing `/jrun` with the correct directory:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample3a
```

If you are using a third-party JDBC driver you must enter the following command, being sure to provide the correct path for the driver:

```
bash$ export JDBC_DRIVERS=/path/driver_name
```

Then enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in the second command prompt window (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample3a
bash$ make run
```

Now as you use the sample you will see that values are persisted between sessions.

Using the instance.store

To run Sample 3 using `instance.store` rather than a database, comment out the `ejipt.*SQL*` properties from the `Balance.properties` file. Be sure to leave the `ejb.containerManagedFields=_id,_value` property in place, telling the server that the `_id` and `_value` fields are to be managed by the container. Then in the `deploy.properties` file uncomment the `ejipt.storeName=default` line and comment out the `ejipt.jdbcSources=source1` line. After making these changes, you must issue `make jars`, `make deploy`, `make standalone`, and `make run` to run the sample (remember to use `makew` on Windows).

CHAPTER 8

Transactions

Contents

Sample 4a - Distributed Transactions

Sample 4a illustrates the use of distributed 2-phase commit transaction management using `server1` and `server2`. The sample also uses both client demarcated, and container managed transactions as illustrated by the implementations of the save and `spend` methods in `EjbClient.java` and `BalanceBean.java`.

First open the `Balance.properties` files and note the following properties:

- `save.ejb.transactionAttribute=tx_mandatory` indicates that a transaction must already be running when the save method is called. In this sample, the `EjbClient.java` program's save method manages the transaction.
- `spend.ejb.transactionAttribute=required` indicates that the EJB engine will start a transaction if a transaction does not currently exist when the `spend` method is called.

Now review the `deploy2.properties` file in `/jrun/samples/sample4a`. The `deploy2.properties` file will be used by `server2`. Notice the following entries:

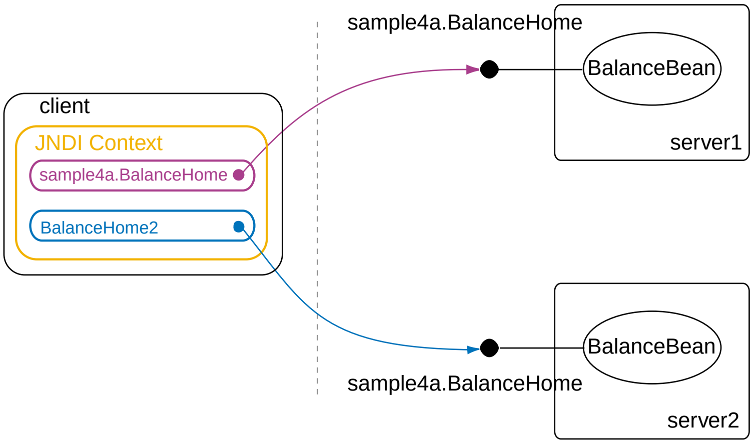
```
sample4a.BalanceHome.maxValue=1000
sample4a.BalanceHome.minValue=-1000
```

These 2 properties specify a valid range for balances that cannot be less than -1000 and cannot exceed 1000 in `server2`. If the minimum or maximum is hit, an exception will be thrown. `Server2`'s `BalanceBean` checks the balance against the values set in the properties file. Hitting the limits in `server2` will result in an exception. This exception will force `server1` to also roll back the transaction.

For more information, refer to the *Developing Applications with JRun* manual.

Customization of the JNDI Context

First we will customize the JNDI context to make our calls simpler and easier to understand. Go to `/jrun/samples/sample4a/client` and open `EjbClient.java`. You can see in the `login` method where the client sets a `context` reference for the server listening on port 2323 (`server1`). The client then sets a *context* reference for the server listening on port 2324 (`server2`) and binds `sample4a.BalanceHome` to `BalanceHome2`. This is done as a convenience to easily differentiate sample4a.BalanceHome for 'server1' and `sample4a.BalanceHome` for 'server2' as illustrated in the following diagram.



Client Demarcated Transactions

Next, take a look at the save method in `EjbClient.java`. The save method will take the responsibility for updating the balance in both `server1` and `server2`. Notice that save creates 2 instances of `Balance`, one associated with each server, and then does a `transaction.begin`. This is required as a result of the `save.ejb.transactionAttribute=tx_mandatory` entry in the `Balance.properties` file.

The `EjbClient.save` method then calls `save(amount)` for both instances of `Balance`. If either call results in an exception, the transaction is rolled back and the balance will not be updated on either server, otherwise `transaction.commit` is called. Therefore, if the balance minimum or maximum in `server2` is hit, `transaction.rollback` is called. As a result `server1` and `server2` will remain in sync.

Implicit Transactions

Now examine the spend method in `EjbClient.java`. Whereas the `EjbClient.java` save method called `BalanceBean.save` for each server, the `EjbClient.java` spend method calls `BalanceBean.spend` once. That is because `BalanceBean.spend` looks for another registered `BalanceBean`. If it finds one, it makes a call to that `BalanceBean.spend` method.

Next take a look at the spend method in `/jrun/samples/sample4a/ejbeans/BalanceBean.java`. The spend method updates its balance, but it is also responsible for updating the balance in `server2` as illustrated with the following snippet:

```
if (_balance2 != null)
{
    _balance2.spend(value);
}
```

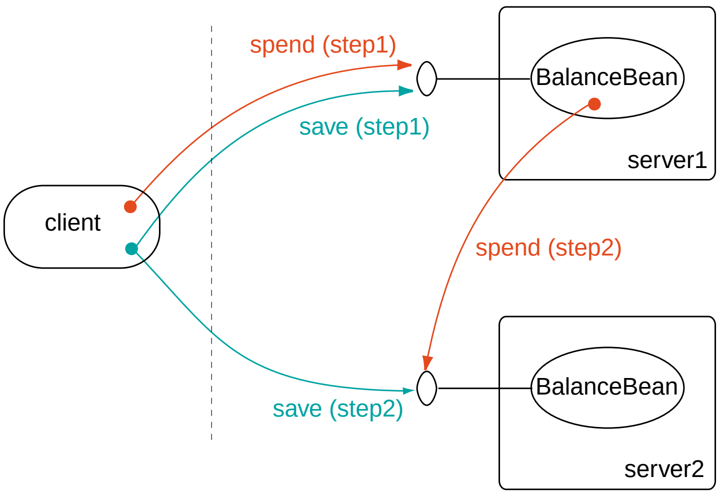
To understand how server1 knows about server2, first review the BalanceBean.connect method. You will see the following:

```
try
{
    final Properties environment = _context.getEnvironment();
    final String host = environment.getProperty
        (EjiptProperties.CLASS_SERVER_HOST);
    final int port = Integer.parseInt
        (environment.getProperty("balance2Port"));
    final Properties properties = new Properties();
    properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
        "allaire.ejipt.ContextFactory");
    properties.setProperty(Context.PROVIDER_URL,
        "ejipt://" + host + ":" + port);
    final BalanceHome home = (BalanceHome)(new InitialContext
        (properties)).lookup("sample4a.BalanceHome");
    _balance2 = home.create(123);
}
catch (NumberFormatException format)
{
    // no port info, connecting is not necessary
}
catch (Exception exception)
{
  ResourceManager.getLogger().logException
      ("Failed to contact other server", exception);
}
```

This code is checking the associated deploy.properties file for a property with the name balance2Port. If you look at deploy.properties in /jrun/samples/sample4a you will find the following entry:

```
sample4a.BalanceHome.balance2Port=2324
```

It then calls getBalance2, which returns a reference to the BalanceHome in server2. Now if you look at the BalanceBean.getBalance2 method you will see that it simply obtains a reference to the BalanceHome for the server listening on port 2324 (previously referred to as server2).



Multiple Server Instances

The `make` files use the `ejipt.ejbDirectory` property to specify `/deploy` and `/runtime` directories for the second server instance. If you have run other samples, be sure to delete the `instance.store` from the `/runtime` directory before running this sample.

To run the sample, open a command prompt and start a shell. For this sample you will need to start RMID. Set your environment variables and enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy2
bash$ make start2.
```

The `make deploy2` and `make start2` commands deploy and start both servers. Now start a client by issuing a `make run`. When you are finished with the sample issue a `make stop2` to stop both servers. Be sure to delete the `/jrun/servers/default/runtime/instance.store` before moving on to the next sample.

Sample 4b - Transactions and CMP

Sample 4b illustrates container managed persistence and uses the transaction management features of Sample 4a. For a complete description of the functionality, see the description for Sample 4a.

This sample uses a relational database. (For the schema, see Sample 2 in Chapter 6). Also be sure to set the database properties in the `deploy.properties` file.

To run the sample, open a command prompt and start a shell. (Be sure to start RMID since this sample runs in fail-safe mode.) Now go ahead and run the sample by entering the following commands, replacing `/jrun` with the correct directory (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample4b
bash$ make jars
bash$ make deploy2
bash$ make start2
```

The `make deploy2` and `make start2` commands deploy and start both servers. Now start a client by issuing a `make run`. When you are finished with the sample issue a `make stop2` to stop both servers.

CHAPTER 9

Object Management

Contents

Overview

Sample 5 focuses on scalability of the server, customization opportunities, and the handling of large numbers of EJB objects.

Note Due to the different performance and garbage collection (GC) characteristics of the stock JDK vs. HotSpot, the results of the following samples will vary.

Sample 5a - Dynamic Object Release

Using an auction paradigm, Sample 5a illustrates the dynamic release of entity objects and overall resource management. As with any auction, there are Bidders that make Bids on available Products.

Products keep a reference to the best Bid. Products can be sold for the best (highest) Bid after a minimum of four Bids have been received. Bidders track their own best bargain; the product that was purchased for the lowest price. However, since product selection and price are randomly generated, it is not guaranteed that all Bidders will successfully purchase a product.

Once you start the sample, the server will create 10,000 Bidders and 10,000 Products. The number of Bidders and Products can be easily customized by changing the following properties in the `Manager.properties` file:

```
numProducts=10000
numBidders=10000
```

The client will begin generating random Bids on Products. Once a batch of Bids have been generated, they are sent to the server for processing.

Bids are processed by the Manager in batches. For each batch the Manager applies each Bid in the batch to the associated Product, then goes through all of the Products to determine which have received enough Bids (at least 4) and can be sold. Those Products that have at least 4 Bids are sold.

To view the code that implements this logic, open the following files in a text editor:

- `/jrun/samples/sample5a/ManagerBean.java`
- `/jrun/samples/sample5a/BidBean.java`
- `/jrun/samples/sample5a/BidderBean.java`
- `/jrun/samples/sample5a/ProductBean.java`

Now start the sample. Be sure to change any necessary host information in the `deploy.properties` file. Enter the following commands, replacing `/jrun` and `hostname` as necessary for your environment (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample5a
```

Enter the following commands to deploy the beans and start the server (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window:

```
bash$ make go host=hostname size=1000
```

The `size=1000` parameter indicates the batch size, the number of bids to be processed in a batch. Changing this parameter will cause batches to be processed more or less frequently for a given auction.

The server will process about 50,000 Bids by running the sample with 10,000 Bidders and 10,000 Products, bringing the total processed entity objects to approximately 70,000. When all products have been sold, the Manager goes through all of the Bidders to determine the time of the best bid. If there was no best bid, then the count of Bidders with no purchases is increased by 1.

De-referenced EJB objects (Bids) are continuously garbage collected during the exercise. The Bids left active will likely be the ones referenced by the Bidders through the purchased products. During the reporting phase of the sample, these active Bids do not have to be reloaded, thus minimizing the number of required `ejbLoads()`.

The sample does not use any databases for persistence. However, it is a simple exercise to add persistence using either CMP or BMP. Since Bids are generated very frequently, indexing Bid tables is not recommended. This restriction, of course, increases the desirability of minimal `ejbLoads()` even further.

The sample was not designed for the client to be restarted. If you want to rerun the sample you must restart the server to ensure a clean environment.

Sample 5b - Custom RMI Sockets

Sample 5b demonstrates customizing RMI sockets in standard ways by providing a `ServerSocket` with a customized `backlog` parameter. This sample could be easily extended to provide custom streams as well as SSL sockets. Third-party standard customizations or products may be used just as easily.

Before starting the sample be sure to change any necessary host information in the `deploy.properties` file. Enter the following commands, replacing `/jrun` and `hostname` as necessary for your environment:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample5b
```

Now enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make go host=hostname count=10
```

The `backlog` parameter for `ServerSockets` defaults to 50. If a connection indication arrives when the queue is full, the connection is refused. The `count=10` indicates the number of simultaneous connections to make to the server.

If you now rerun the sample with more simultaneous connections than the backlog queue can accommodate you will encounter errors:

```
bash$ make go host=hostname count=150
```

To prevent these errors go to the `Product.properties` file and uncomment the 2 socket factory properties and then enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make go host=hostname count=350
```

You will see that the backlog queue has been expanded to handle the 350 simultaneous connections. Although it is unlikely that the server would encounter 350 connection requests arriving at the exact same moment, you can use this type of customization to prevent connections from being refused when there are spikes in activity.

Sample 5c - Large Enumerations

This sample demonstrates returning a large `Enumeration` of Bids to the client. The client then iterates through the bids to get the price. The sample illustrates how objects are garbage collected once a client is no longer referencing them.

Review the `ejbFindAll` method and the `KeyEnumerator` inner class in `BidBean.java`. Also review the `EjbClient.java` class and notice how the `run` method includes code that calls the EJB's `FindAll` method to return an `Enumeration` of Bids.

Before starting the sample be sure to change any necessary host information in the `deploy.properties` file. Enter the following commands, replacing `/jrun` and `hostname` as necessary for your environment:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample5c
```

Now enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make go host=hostname size=1000
```

The `size=1000` parameter indicates the size of the `Enumeration` to be returned by the server to the client.

CHAPTER 10

Messaging

Contents

Overview

Sample 6 highlights the features of the Java Message Service (JMS). The features illustrated include point-to-point (queues) and publish/subscribe (topics) messaging.

Note The Multicast Time-To-Live property (`jms.multicast.ttl`) is set to zero, which will prevent UDP packets from being forwarded to remote clients. To use the following samples with remote clients, set the `jms.multicast.ttl`, `jms.multicast.port`, and `jms.multicast.groupAddress` properties in the `deploy.properties` file to the correct values for your environment.

These samples use the `instance.store` for persisting messages. You should remove the `instance.store` from the `/runtime` directory before running each sample.

For more information on using JMS, refer to *Developing Applications with JRun*.

Sample 6a - Point-to-Point

Sample 6a illustrates point-to-point or queue-based messaging services. The sample demonstrates synchronous messaging where messages are written to a queue, the queue is then polled for messages. Asynchronous messaging, where listeners register to automatically receive messages, is also demonstrated.

Start by reviewing the `/sample6a/deploy.properties` file. To enable messaging, set the `ejipt.enableMessaging` property to `true`. This notifies the EJB engine to load the messaging related beans. If this property is not set to `true`, messaging will not work.

Also notice the following property setting:

```
default.MessageQueueHome.ejb.enterpriseBeanClassName=ejbeans.QueueBean
```

This tells JRun that `ejbeans.QueueBean` is to be used for persisting messages rather than the `default.MessageQueueBean`. Taking a look at `/sample6a/ejbeans/QueueBean.java` you will see that it implements the `onAdding` and `onRemoved` methods. The `onAdding` method is called just prior to adding the message to the queue whereas `onRemoved` is called just after the message is removed from the queue. This enables the sample to write entries to the JRun log file. Your applications can optionally extend `MessageQueueBean` to implement similar functionality, but it is not typically required.

There are two client side applications: `Sender` and `Receiver`. First review `Sender` by going to `/sample6a/client/Sender.java`. `Sender` accepts four arguments, as described in the following table.

Sample 6a Sender Arguments

Parameter	Value
host	Specifies the host name of the server or 'localhost' if the server and `Sender` are running locally.
queue name	Identifies the queue that messages will be sent to.
mode	Indicates whether Sender will automatically generate a series of messages. Values: `manual` or `auto`.
name	Specifies the name that identifies the sender.

Reviewing `Sender.java` you will see that `Sender` must first get a reference to a `QueueConnectionFactory` and with that get a `QueueConnection`. With the connection established it can go ahead and create a `QueueSession`. Only then can it begin to send messages. `Sender` also creates the actual message queue using the queue name parameter.

To generate and send an actual message, `Sender` calls `Message.setText` with the text, and then calls `QueueSender.send` with the message, delivery mode, priority and the time interval until expiration.

Now go to `/sample6a/client/Receiver.java` to see how messages can be retrieved. `Receiver` accepts three arguments, as described in the following table.

Sample 6a Receiver Arguments

Parameter	Value
host	Specifies the host name of the server or 'localhost' if the server and `Receiver` are running locally.
queue name	Identifies the queue from which messages will be retrieved.
mode	Indicates if the receiver will manually retrieve messages from the queue (synchronous) or if it will register itself as a listener to automatically receive messages (asynchronous). Values: 'manual' or 'auto'.

`Receiver` must also get a reference to a `QueueConnectionFactory` and with that get a `QueueConnection` and then create a `QueueSession`. This time, rather than creating an sender, a `QueueReceiver` is created. When mode is `auto`, `Receiver` registers as a `listener` for the queue (asynchronous). Otherwise the queue is checked each time the Enter key is pressed (synchronous).

To run the sample, start by bringing up the server by entering the following commands. Ignore the `No beans found in jar(s)` message Remember to use `makew` on Windows:

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Next open a new shell and start the Sender by entering the following command (remember to use `makew` on Windows):

```
bash$ make sender host=localhost queue=cat mode=manual name=fluff
```

You will see output similar to the following:

```
Type message to send or ’quit’ to exit, then press <ENTER>
```

Now type in some text and hit Enter. You will see the following:

```
Sending: [delivery: non-persistent, priority: default, from: fluff]
Content: <your text here>
Type message to send or ’quit’ to exit, then press <ENTER>
```

You can prefix messages with `:dp` to specify that DeliveryMode is persistent when sending messages:

```
:dpmessage text
```

In this case the message will survive a server shutdown. Messages are persisted by `QueueBean` using the `instance.store`. This can be tested by sending messages, stopping and starting the server and then starting `Receiver`. Keep in mind that the message’s default time-to-live is set to 5 minutes in `Sender`, therefore a message could expire before being delivered.

You can also set the message’s priority by prefixing messages with `:pX` where X=0 through 9 with 9 being the highest priority. Messages with higher priority are delivered before messages of a lower priority.

```
:p9message text
```

You can try this by sending several messages with varying priority. Then do a series of receives, you will receive the messages in order of highest to lowest priority.

By starting the sender with `mode=auto`, a series of messages will automatically be generated and sent out (remember to use `makew` on Windows):

```
bash$ make sender host=localhost queue=cat mode=auto name=fluff
```

Now open a new shell and start the `Receiver` by entering the following command (remember to use `makew` on Windows):

```
bash$ make receiver host=localhost queue=cat mode=manual
```

You will then see the following output:

```
Press <ENTER> to receive message or enter ’quit’ to exit
```

Now press Enter to retrieve your messages:

```
Received: [delivery: non-persistent, priority: -1, from: fluff]
Content: <your text here>
Press <ENTER> to receive message or enter 'quit' to exit
```

For the `Receiver` you can enter `:wXXX` on the command line where XXX represents the number of seconds the receiver will wait for new messages. The `Receiver` will wait for the number of seconds specified or until a message arrives, whichever comes first.

By starting `Receiver` with `mode=auto`, it will receive asynchronous messages, no polling is necessary. In this case `Receiver` must register as a `listener` of the queue. Any messages that may already be on the queue at the time `Receiver` registers as a listener will not be sent to `Receiver` since `Receiver` was not registered as a `listener` at the time the message was produced.

Sample 6b - Publish/Subscribe

Sample 6b demonstrates publish/subscribe (topic-based) messaging support. With topics, all *Subscribers* who have registered will receive the message provided that the message has not expired. Messaging priority and persistence can be specified using the same prefixing as described in Sample 6a.

Examine the `/sample6b/client/Publisher.java` file's `run` method to see how the sample uses the `publish` method to publish messages to the topic. Review the `/sample6b/client/Subscriber.java` file's `onMessage` method to see how the sample displays received messages.

To run the sample, bring up the server by entering the following commands. Ignore the `No beans found in jar(s)` message. Remember to use `makew` on Windows:

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now you must start some Subscribers so that they will be registered as listeners before Publisher begins to send messages. A registered listener will only receive messages that are new to the topic. Messages already in the topic will not be sent to a newly registered listener.

Open a new shell and enter the following command. Do this for at least two Subscribers (remember to use `makew` on Windows):

```
bash$ make subscriber host=localhost topic=dog mode=auto
```

Next open a new shell and start a `Publisher` by entering the following command (remember to use `makew` on Windows):

```
bash$ make publisher host=localhost topic=dog name=spot
```

As you send the messages you see that the Subscribers automatically receive the messages. When using auto mode Subscribers will only receive messages that are published while the subscriber is active.

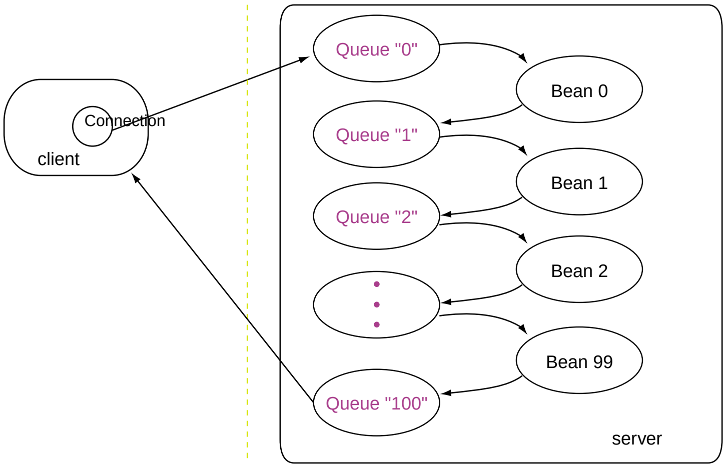
When starting subscriber in manual mode you can retrieve any messages that are currently in the message buffer. To set the size of the buffer set the `jms.messageCapacity` property.

Topics can be hierarchical in nature. To see how this works take a look at `/sample6b/ejbeans/TopicBean.java`. There you will see the method `ejbFindSuperTopics`. This method looks for other topics that have the same prefix. For example if you have the topics dog and dog.lab then dog is the super topic and dog.lab is the subtopic. Therefore subscribers of dog will also see dog.lab messages but dog.lab subscribers will only see dog.lab messages and will not see dog messages. Subscribers of dog.lab would however see messages to dog.lab.black. `TopicBean.java` can be easily customized to follow topic conventions specific to your environment.

Sample 6c - EJB Integration

Sample 6c demonstrates the ability for EJBs to fully participate and interact with the JRun JMS implementation. The client will produce an initial message that will result in a chain of messages being produced and consumed by beans on the server with a final message being consumed by the originating client.

To understand how this works, start by examining `Client.java` in `/sample6c/Client.java.` In the constructor method you will see where, after establishing a connection and a session, the client will create 100 queues named “1” through “100”. It will also create 100 instances of the `ListenerBean` entity bean and finally will register itself as a listener for the queue named “100”.



In the `Client.run` method a message is sent to queue “0” and then the method waits until the `onMessage` method is executed. Notice that `Client` implements the `MessageListener` interface and must therefore implement the `onMessage` method. The `onMessage` method will be called when a message is forwarded to queue “100”.

To see how the chaining is implemented let’s review `Listener`. First go to `/sample6c/ejbeans` and open the `Listener.properties`. Notice here that the `ejipt.maxContexts` property is set to “10”, meaning that the number of active bean instances cannot exceed 10. Therefore instances will be activated and passivated as necessary while the sample is running. The `instance.store` is used for persistence.

Next lets take a look at `ListenerBean.java`. In the `setEntityContext` method we check to see if this is the first instance, if so we bind the connection in the JNDI context so that all instances of `Listener` can use the same connection.

Since the `Listener` object (`Listener.java`) extends `MessageListener`, we must provide an implementation of the `onMessage` method in `ListenerBean.java`. This method is responsible for forwarding the message to the next queue.

Now run the sample. Enter the following commands to start the server (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start the client by entering the following command (remember to use `makew` on Windows):

```
bash$ make go host=localhost
```

When running the sample you will see the following output on the server:

```
>[object:0] forwarding message...
>[object:1] forwarding message...
>[object:2] forwarding message...
.
.
.
>[object:99] forwarding message...
>
```

And you will see the following output on the client:

```
Received: 100
```

When the chain is complete, the `Client.run` method removes the `Listener` instances from the `instance.store` and closes the `Receiver`, `Sender`, `Session`, and `Connection`.

CHAPTER 11

Advanced Beans

Contents

Overview

Sample 7 illustrates the use of entity beans, stateful session beans and stateless session beans all working in conjunction with one another. This sample also demonstrates deadlock exception handling and autocallers.

This sample uses a command line entry to start a client process. The sample is designed to demonstrate how different beans can interact with one another. Information is set at runtime through properties and command line arguments. Output is written to the console window. Data is persisted to the `instance.store`. This application can easily be used as a framework for real world distributed applications.

The functional description of the business logic is fairly straightforward. Banks loan money to certain Customers and then those Customers must repay the Loans within a preset time period. If the Loan is not repaid during the allotted time period the Loan defaults.

Banks earn income by charging interest on the Loans to Customers. The types of Banks range from conservative to risky. Customers have some initial worth (inheritance) and this worth accrues over time as a result of income. The income amount is determined as a percentage of their current worth. Customers may borrow additional money to temporarily increase their worth, but can only have one loan outstanding at any time. The types of Customers range from conservative (those that borrow less) to risky (those that borrow more). Customers also have credit ratings that start out neutral and are adjusted as Loans are either paid or defaulted.

There is a `Web` entity bean that represents the universal source of information. `Web` can be browsed for the list of Banks, as well as for both the current interest rate and income rate. Customers use `Web` to locate Banks. Banks use `Web` to determine the current interest rate. The interest rate and income rate are set in the `Web.properties` file.

`CustomerSession` is a stateful session bean manages a session for each `Customer` entity bean and updates the Customer's worth at defined intervals. `Loan` is a stateful session bean that represents each loan for the duration of its existence, once a `Loan` instance is paid or defaulted the loan instance expires and is removed. `Calculator` is a stateless session bean used to calculate the amount of each installment for a specific `Loan`.

Beans

The following table lists the beans used in this sample. To review the JavaDocs for this sample go to `/jrun/samples/sample7a/docs`.

Sample 7 Table Schema	
Bean	**Type**
Bank	Entity
Customer	Entity

Sample 7 Table Schema (Continued)	
Bean	**Type**
Web	Entity
Loan	Stateful Session
CustomerSession	Stateful Session
Calculator	Stateless Session

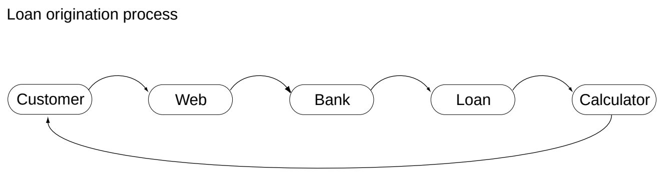
Process

When starting the server, the number and types of banks defined in the `deploy.properties` file are created. The server also initializes the Web with the current interest and income rates.

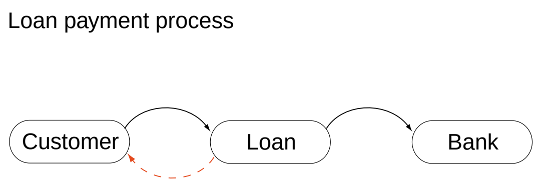
When starting a client, arguments are passed that indicate the host name and port, and the number of `Customers` along with their initial worth. `Customers` randomly begin asking for loans and `Web` randomly selects banks that can provide the loans.

The `Customer` then asks the `Bank` for a `Loan`, the `Bank` in turn determines if it wants to loan money to that `Customer`. The decision criteria is based on the availability of `Bank` funds and the `Customer`'s credit rating. Once a `Loan` is created by the `Bank`, the `Customer` uses the calculator to determine the amount of the installment payments.

Once a loan application has been approved, the created `Loan` must be paid back.



The `Customer` sends the payment to the `Loan`, the `Loan` then passes the payment amount to the `Bank` to be added back into the `Bank`'s available funds. For each `Loan`, the `Customer` is scheduled to make 10 payments. If the `Loan` defaults, the `Customer`'s credit rating is adjusted (by the `Loan`) before the `Loan` is removed.



Deadlocks

Sample 7 was designed to illustrate a deadlock-prone situation and how the EJB engine detects and breaks deadlocks. The `Loan` lives only for a certain fixed period of time. If the `Customer` does not pay the `Loan` in time, the `Loan` is defaulted, even when the `Customer` is still paying installments. If the `Loan` has not been fully repaid at the time it expires, the `Loan` downgrades the `Customer`'s credit rating in `Loan.ejbRemove`. However, if the `Loan` is trying to downgrade the credit rating at the same time the `Customer` is trying to pay the `Loan`, a deadlock situation may occur.

There are two places in the sample where deadlocks are managed. `DeadlockExceptions` are declared as checked exceptions in the throws clause of the `LoanBean.payInstallment` method. This forces the EJB engine to throw deadlock exceptions unchanged. Without these declarations the exceptions would be wrapped into `java.rmi.RemoteExceptions`.

`DeadlockExceptions` are caught in the `CustomerBean.payInstallment` method and ignored. As a result, a call attempting to update the `Customer`'s credit rating will eventually go through so the `Loan` can be defaulted and removed. The next time the customer tries to pay the `Loan`, a `NoSuchObjectException` is caught and the `Customer` now knows the `Loan` has expired and ceases payments.

Sample 7a - Complex Processing

Before starting Sample 7a, take a moment to review the source files located in `/jrun/samples/sample7a`. To begin, go to the `/sample7a/ejbeans` directory and open the `Web.properties` file. Set the interest rate and income rate, if you so desire. Also be sure to change any necessary host information in the `deploy.properties` file.

Next, start up a shell as described in Sample 1 (described in Chapter 5), set the `JRUN_HOME` variable and change to the `/jrun/samples/sample7a` directory.

Now go ahead and start the demo by entering the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window. To start the client enter the following command (remember to use `makew` on Windows):

```
bash$ make simulate host=localhost first=0 last=10 inheritance=1000
```

This command will cause the server to create 11 concurrent Customers with id's 0 through 10. If you have changed the server port to something other than 2323, for example 2324, you would enter the following command (remember to use `makew` on Windows):

```
bash$ make simulate host=hostname:2324 first=0 last=10 inheritance=1000
```

Now that you have run the simulation, try running it again, this time entering additional customers or changing the inheritance amount.

Running with 100 Customers will result in 1000 Loans and approximately 10,000 payments (depending on the ejb.sessionTimeout setting for Loan durations in `Loan.properties`). Try running with 1000 customers. Also try running an additional client by opening another command window and entering the following (remember to use `makew` on Windows):

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample7a
bash$ make simulate host=localhost first=200 last=210
inheritance=1000
```

Review the Sample 7a JavaDoc style documentation for additional information, it can be found in `/jrun/samples/sample7a/docs`.

Sample 7b - Complex Processing with BMP

Sample 7b adds bean managed persistence using a relational database to Sample 7. You will need access to a relational database to run Sample 7b.

Sample 7b has been tested with Oracle using the Thin JDBC driver. It has also been tested with SQLServer using the stock JDBC/ODBC driver.

The SQL scripts included in the `/jrun/samples` directory can be used to load the database schemas used by Sample 7b. Both Oracle and SQLServer versions are provided. The following table defines the schema used in Sample 7b:

Sample 7b Schema

Table	Field	JDBC Type	Rules
account	id	INTEGER	NOT NULL
	value	INTEGER	NOT NULL
bank	name	VARCHAR (16)	NOT NULL
	type	INTEGER	NOT NULL
	funds	FLOAT	NOT NULL
	loans	FLOAT	NOT NULL
customer	name	VARCHAR (16)	NOT NULL
	password	VARCHAR (16)	NOT NULL
	type	INTEGER	NOT NULL
	rating	INTEGER	NOT NULL DEFAULT 0
	worth	FLOAT	NOT NULL

Sample 7b Schema (Continued)

Table	Field	JDBC Type	Rules
	handle	VARBINARY (128)	NULL DEFAULT NULL
	installment	FLOAT	NOT NULL

To begin this sample, go to `/jrun/samples` and open the appropriate `.sql` file for your database. These files contains table definitions as well as `create_customer`, a stored procedure for creating customers in the database. You must create the tables and install the stored procedure to run the sample.

Now review `deploy.properties` in `/sample7b`. Notice there are properties defining the JDBC source that are currently set up to use the JDBC/ODBC bridge. Be sure to update the `ejipt.jdbcSources`, `ejipt.sourceURL`, `source1.ejipt.sourceUser`, and `source1.ejipt.sourcePassword` property settings as necessary for your environment. In particular be sure to set the @your_host in `source1.ejipt.sourceURL` if you are using a third-party driver.

Now review `BankBean.java` in `/sample7b/ejbeans`. You will notice SQL related references in the `ejbPostCreate, ejbLoad`, and `ejbStore` methods.

To begin the sample, go to the `/sample7b/ejbeans` directory and open the `Web.properties` file. You may set the 'interest rate' and 'income rate'. Be sure to change any necessary host and JDBC information in the `deploy.properties` file.

Now start the sample. Enter the following commands, replacing `/jrun` as necessary for your environment:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample7b
```

If you are using a third-party JDBC driver, you must enter the following, being sure to provide the correct path for the driver:

```
bash$ export JDBC_DRIVERS=/path/driver_name
```

Now enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make simulate host=hostname first=0 last=10 inheritance=1000
```

Sample 7c - Complex Processing with CMP

Sample 7c takes the applications used in Sample 7 and adds container managed persistence to a relational database.

Sample 7c uses the same database schema as Sample 7b. For the schema as well as additional information on JDBC drivers, see Sample 7b.

If you did not create the tables described in Sample 7b, then you should do so now. Sample 7c uses the same tables as Sample 7b. Also be sure to set the JDBC properties in `/sample7c/deploy.properties`.

If you will now take a look at `Bank.properties` in `/sample7c/ejbeans/`, you will see the CMP properties. The first property is `ejb.containerManagedFields`, telling the container the fields to be managed by the container. The `Customer.properties` file also contains CMP properties.

The `ejipt.*SQL*` properties define how the data in the bean is stored and retrieved. You will notice that in this sample there are properties for `create`, `load` and `store`. These properties are used by the corresponding ejb methods to properly manage the bean instance's state.

Now let us review the `BankBean.java` in `/sample7c/ejbeans`. You will notice less code than in the corresponding Sample 7b `BankBean.java`. Notice in particular the `ejbPostCreate`, `ejbLoad`, and `ejbStore` methods.

To begin the sample, go to the `/samples/sample7c/ejbeans` directory and open the `Web.properties` file. Reset the interest rate and income rate, if desired. Be sure to change any necessary host and JDBC information in the `deploy.properties` file.

Now start the sample. Enter the following commands, replacing `/jrun` and `hostname` as necessary for your environment:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/samples/sample7c
```

If you are using a third-party JDBC driver, you must enter the following, being sure to provide the correct path for the driver:

```
bash$ export JDBC_DRIVERS=/path/driver_name
```

Now enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now start up the client in another command prompt window (remember to use `makew` on Windows):

```
bash$ make simulate host=hostname first=0 last=10 inheritance=1000
```

Prepared Statements

Sample 7c uses prepared statements for accessing the database. Some databases, such as MS SQLServer with the stock JDBC/ODBC driver, will get 'invalid ResultSet' errors. To prevent this error, add the following statement to the `deploy.properties` file:

```
ejipt.disableStmtPool=true
```

CHAPTER 12

Using EJB with Servlets

Contents

Overview

Servlets are an excellent way to communicate with clients when your clients are outside a firewall. This sample demonstrates accessing beans from a servlet.

Sample 9a

Before you begin, review the `.java` files in the `sample9a/webapp/WEB-INF/classes` directory. They give you an idea of the code you add for servlets to communicate with an EJB. To run this example, you use the following `make` (or `makew`) options:

- `make jars`: Compiles EJB files and creates a .jar file.
- `make deploy`: Deploys the EJB.
- `make war`: Compiles the servlets and creates a `.war` file.
- `make wardeploy`: Deploys the `.war` file.
- `make startup`: Starts the JRun default server.

If you have run other samples, be sure to delete the `instance.store` from the `/runtime` directory before running this sample.

To run sample 9a, open a command prompt and start a shell. Set your environment variables and change to the `sample9a` directory. Enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make war
bash$ make wardeploy
bash$ make startup
```

Open your favorite browser and be sure it is set to accept cookies. Now point to `http://`*`hostname`*`:`*`portnumber`*`/sample9a`. You should see a login screen similar to the Java applications you have seen in the other samples. Login using ‘chief’ and ‘pass’ and enter transactions. This sample writes output to the `default-event.log` file.

The servlet creates one cookie per client, therefore if you start a second browser on the same machine it will have the same identity.

To stop the server, enter `CTRL+C`.

CHAPTER 13

JDK 1.1 Clients

Contents

Sample 10a - Using JDK1.1 Clients

Sample 10a demonstrates using the EJB engine with JDK 1.1 and Java 2 clients concurrently. There are extra steps involved when using JDK 1.1 clients, these steps are necessary because JDK1.1 does not support RMI class loading from multiple URLs.

First, go to the `deploy.properties` for Sample 10a and set the server name to the host's name. Also notice the `ejipt.isCompatible=true` property, this property notifies the EJB engine that RMI skeletons for 1.1 clients must be generated.

Now open a command prompt and start a shell. Set the JRUN_HOME environment variable, change to the `/sample10a` subdirectory and enter the following commands (remember to use `makew` on Windows):

```
bash$ make jars
bash$ make deploy
bash$ make standalone
```

Now connect to the client machine and create a directory named `/sample10a`. Copy the following files from the server to the `/sample10a` directory on the client:

- `/jrun/samples/sample10a/sample10a_client.jar`
- `/jrun/servers/default/runtime/ejipt_exports.jar` - includes the stubs
- `/jrun/lib/ejipt_client.jar` - JRun standard extensions
- `/jrun/lib/ext/ejb.jar` - Java standard extensions
- `/jrun/lib/ext/jta.jar` - Java standard extensions
- `/jrun/lib/ext/jndi.jar` - Java standard extensions

Now open a command prompt on the client and change to the `/sample10a` directory you just created. Set your classpath as follows:

```
> set CLASSPATH=sample10a_client.jar;ejipt_exports.jar;
  ejipt_client.jar; ejb.jar;jta.jar;jndi.jar;C:\jdk\lib\classes.zip
```

The following command will start the client application. Be sure to set `host` to the name you specified in the `deploy.properties`.

```
> java Client1_1 host chief pass
```

Running the client with the saver1 or spender1 user ids will demonstrate additional authentication. The client connects to the server and calls the `save` method to save 100 with a repeat of 10 and then calls the `spend` method to spend 100 with a repeat of 10. You may also start up the application for Java 2-based clients using `make run` (remember to use `makew` on Windows).

CHAPTER 14

Make Files

Contents

Using Make Files

This chapter describes in detail the make files provided with the samples. The discussion uses Sample 1a for the examples.

If you prefer, you can enter the commands directly into a command prompt window rather than using `make` files. To review the commands see the Deploying Beans chapter in *Developing Applications with JRun*.

Make and makew

JRun includes separate `make` files for UNIX and Windows. The UNIX `make` files are named `make` and use the GNU `make` utility. The Windows make files are named `makew` and are `.bat` files. Anytime a `make` command appears in this manual, Windows users should substitute `makew`.

Understanding the make files

This section describes the contents of the `make` files, used to run the samples under UNIX. For information on running the `make` files under Windows (instead of `makew`), see “Note to Cygnus users” on page 76.

The names of the bean’s home and remote interfaces, and the bean’s implementation must be specified as illustrated in `/jrun/samples/sample1a/ejbeans/Makefile`. All of your bean files must be included using the following format:

```
sources = $(addprefix ejbeans/, \
RemoteName.java\
BeanName.java\
HomeName.java\
)
```

Client related java files must also be specified. `/jrun/samples/sample1a/client` illustrates this, it contains all of the names of client related files. Add your client files using the following format:

```
sources = $(addprefix client/, \
ClientUI.java\
MainPanel.java\
)
```

The bean’s home and remote interface classes must also be specified, as illustrated in `/jrun/samples/sample1a/Makefile`. Include your bean’s home and remote interfaces using the following format:

```
ejb_clients = \
ejbeans/RemoteName.class\
ejbeans/HomeName.class
```

Understanding the makew files

This section describes the contents of the `makew` files, used to run EJB samples under Windows.

The names of the bean's home and remote interfaces, and the bean's implementation must be specified, as illustrated in `/jrun/samples/sample1a/ejbeans/makew.bat`. All of your bean files must be included using the following format:

```
@set sources=ejbeans\Balance.java ejbeans\BalanceBean.java
ejbeans\BalanceHome.java
```

Client related java files must also be specified, as illustrated in `/jrun/samples/sample1a/client/makew.bat`. Add your client files using the following format:

```
@set sources=client\ClientUI.java client\EjbClient.java
client\LoginEvent.java client\LoginPanel.java client\MainPanel.java
client\Request.java
```

The bean's home and remote interface classes must also be specified, as illustrated in `/jrun/samples/sample1a/make1.bat`. Include your bean's home and remote interfaces using the following format (in this example, `Balance.class` is the remote interface and `BalanceHome.class` is the home interface):

```
@set ejb_clients=ejbeans\Balance.class ejbeans\BalanceHome.class
```

Using Make Jars

To use the make files, open a command prompt and start a (bash or DOS) shell. Set `JRUN_HOME` to the JRun installation directory, change to your working directory and enter `make jars` as follows:

```
bash$ export JRUN_HOME=/jrun
bash$ cd /jrun/projectpath
bash$ make jars
```

For Windows, start a DOS shell and enter `makew jars` as follows:

```
set JRUN_HOME=/jrun
cd /jrun/projectpath
makew jars
```

Using Make Deploy

The Deploy tool is used to generate the home and remote implementations. It uses the JDK compiler; however you can override that by setting the `ejipt.javac` property in the `deploy.properties` file.

Once you have created the bean's jar file, you can deploy the beans using `make deploy`, forcing all bean implementations to be regenerated (remember to use `makew` on Windows):

```
bash$ make deploy
```

You will see output similar to the following:

```
cd /jrun; java -Djava.security.policy=jrun.policy
-classpath lib/ejipt_tools.jar allaire.ejipt.tools.Deploy
Generating BalanceHomeObject...
Generating BalanceObject...
```

```
Compiling files...
Generating BalanceHomeObject_Stub...
Generating BalanceObject_Stub...
Compiling files...
```

Using Make Redeploy

The `make redeploy` command calls the Deploy tool with the `-redeploy` option. The `-redeploy` option tells the Deploy tool to generate implementations only for those beans that are new or have been updated since the last time the Deploy tool was run (remember to use `makew` on Windows):

```
bash$ make redeploy
```

You will see output similar to the following:

```
cd /jrun; java -Djava.security.policy=jrun.policy
-classpath lib/ejipt_tools.jar allaire.ejipt.tools.Deploy -redeploy
```

Using Make Standalone

The `make standalone` command starts the EJB engine in stand-alone mode using the `.jar` files in the `/deploy` directory. The `.jar` and `.properties` files in the `/deploy` directory are copied to the `/runtime` directory (remember to use `makew` on Windows):

```
bash$ make standalone
```

You will see output similar to the following:

```
cd /jrun; java -Djava.security.policy=jrun.policy -classpath
    lib/ejipt.jar allaire.ejipt.Ejipt
```

Once the EJB engine has completed startup it is ready to handle requests. You can then start your clients and connect to the server.

The `make standalone` command starts the EJB engine using the directories and port settings of the JRun default server. The JRun default server cannot be running when you issue `make standalone`. Running the EJB engine in stand-alone mode allows you to view bean processing in a console window. As an alternative to `make standalone`, restart the JRun default server after running `make deploy`, issue the `make` command for the client process, run the client application, and view results in the log file for the JRun default server (`/jrun/logs/default-event.log`).

Using Make Classes

The `make classes` command compiles modified bean implementations into the `/runtime/classes` directory. After running this command, you must use the `load` command to reload the bean from the `/runtime/classes` directory. Use this command to implement dynamic bean loading (remember to use `makew` on Windows):

```
bash$ make classes
```

Using Make Start

The `make start` command starts the EJB engine in fail-safe mode using the `.jar` files in the `/deploy` directory. The `.jar` and `.properties` files in the `/deploy` directory are copied to the `/runtime` directory (remember to use `makew` on Windows):

```
bash$ make start
```

You will see output similar to the following:

```
cd /jrun; java -Djava.security.policy=jrun.policy -classpath
    lib/ejipt_tools.jar allaire.ejipt.tools.Server -start
```

Once the server has started it is ready to handle requests. You can then start your clients and connect to the server.

Using Make Restart

The `make restart` command starts the EJB engine in fail-safe mode using the `.jar` files previously copied to the `/runtime` directory (remember to use `makew` on Windows):

```
bash$ make restart
```

You will see output similar to the following:

```
cd /jrun; java -Djava.security.policy=jrun.policy -classpath
    lib/ejipt_tools.jar allaire.ejipt.tools.Server -restart
```

Once the server has started it is ready to handle requests. You can then start your clients and connect to the server.

Choosing the Right Makefile

The following are guidelines regarding when to use the most common `make` commands.

Make Files

Makefile	When to use

`make jars`
`makew jars`
	Use this command any time the bean's implementation or interfaces have been modified. Also use if either the bean's properties or default.properties have changed.

`make deploy`
`makew deploy`
	Use this command when you want to generate object implementations for all of your beans. By default the Deploy tool generates implementations of all beans.

`make redeploy`
`makew redeploy`
	This command forces the Deploy tool to generates object implementations only for new or updated beans.

Make Files (Continued)	
Makefile	**When to use**

`make standalone`
`makew standalone`
	This command starts the EJB engine in stand-alone mode using the jar files from the /deploy directory. It copies the bean jars the Deploy tool previously processed, along with the generated runtime.properties file, from the /deploy directory to the /runtime directory. It also copies the object implementation (ejipt_objects.jar) and stubs (ejipt_exports.jar) jars.

`make start`
`makew start`
	This command starts the EJB engine in fail-safe mode using the jar files from the /deploy directory. It copies the bean jars that the Deploy tool has previously processed, along with the generated runtime.properties file, from the /deploy directory to the /runtime directory. It also copies the object implementation (ejipt_objects.jar) and stubs (ejipt_exports.jar) jars. RMID must be started prior to make start.

`make restart`
`makew restart`
	This command starts the EJB engine using the jar and runtime.properties files previously copied to the runtime directory. This implies that the EJB engine cannot be “restarted” without first “starting” it. RMID must be started prior to make restart.

`make classes`
`makew classes`
	This command compiles modified beans into the runtime/classes directory.

`make allclean`
`makew allclean`
	This command removes all of the generated files from the /ejbeans and /client directories.

Using Fail-Safe Mode

Except where noted, EJB samples can be run in stand-alone mode or in fail-safe mode using RMID. The sample descriptions contain instructions for stand-alone mode. However if you prefer to run in fail-safe mode, be sure to start RMID by following the directions below, then replace `make standalone` with `make start` (remember to use makew on Windows). If you prefer to use command line entries rather than `make` files, enter the following commands:

```
> cd /jrun; java -Djava.security.policy=jrun.policy -classpath
  lib/ejipt_tools.jar allaire.ejipt.tools.Server -start
```

To stop the server when running in fail-safe mode enter `make stop` or enter the following command:

```
> cd /jrun; java -Djava.security.policy=jrun.policy -classpath
  lib/ejipt_tools.jar allaire.ejipt.tools.Server -stop
```

JRun will take a moment to shutdown while it performs clean up and finalizes transactions. Once it has stopped, be sure to shut down RMID services by following the instructions below.

Using RMID

RMID provides remote activation of the server. When a client attempts to connect to the server, RMID ensures the server is available by starting the server if necessary. To start RMID use the instructions appropriate for your environment.

RMID on Solaris and Linux

To start RMID on Solaris or Linux enter the following commands:

```
% cd /tmp
% rmid
```

By default, you will receive output messages into the RMID log window. To stop RMID on Solaris enter the following command:

```
% rmid -stop
```

RMID on Windows

To start RMID on Windows open a command prompt window and enter the following commands:

```
> cd \temp
> start rmid
```

A new window will appear with the title rmid.exe. By default, you will receive output messages into the new window. This window will stay open until RMID is stopped.

To stop RMID on Windows enter the following command:

```
> rmid -stop
```

Troubleshooting RMID

RMID creates a log file that is used to automatically restart the server. To start a new instance of the server, the log file should first be deleted. The `log` directory was created in the directory you were in when you started RMID.

If you attempt to start the server in fail-safe mode without first starting RMID, the server will not start up properly. If this has occurred or you receive an exception with the message “Failed to acquire lock, operation aborted” you must reset your environment. To reset your environment and start the server, perform the following steps:

1. Issue a `make stop` or equivalent.
2. Access the `/runtime` directory and remove the `*.id` and `*.lock` files.

3. Be sure to remove any `/log` directories from prior instances of RMID.
4. Issue “`start rmid`”.

Issue `make start` or equivalent.

Note to Cygnus users

You can also use a Windows version of the GNU `make` utility if you prefer. If you are using the Cygnus tools, the following environment variable must be set:

```
MAKE_MODE=UNIX
```

If you installed JRun in the `Program Files` directory and are using the `bash` shell, you will have to use escape characters to handle the space in the name when working in a bash shell. The commands will be as follows:

```
bash$ export JRUN_HOME=”/Program\ Files/Allaire/JRun”
bash$ cd /Program\ Files/Allaire/JRun/samples/sample1a
```

If you installed JRun to a directory on a drive other than the `C:` drive, the commands you enter when working in a `bash` shell will be slightly different. For example, if you installed to “`D:\jrun`”, the commands would be as follows:

```
bash$ export JRUN_HOME=d:/jrun
bash$ cd //d/jrun/samples/sample1a
```

Index